Cicada: Player-Scalable, Fault-Tolerant Secure MultiParty Computation

Chesapeake Large-Scale Analytics Conference

Jon Berry, Nov. 1, 2023

Project Team:

J. Berry (PI), G. Birch, K. Dixon (PM), A. Ganti, K. Goss, C. Mayer, U. Onunkwo, C. Phillips, J. Saia (UNM), T. Shead

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

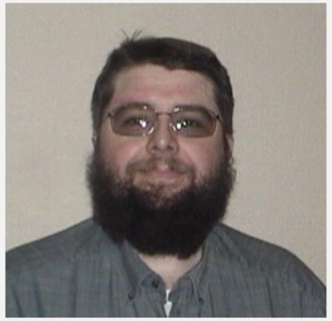
Thanks to Our Multi-Disciplinary Research Team


Jon Berry (PI)

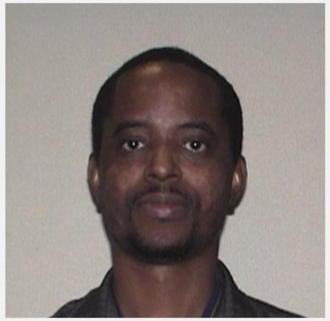

Gabe Birch


Kevin Dixon (PM)


Anand Ganti


Ken Goss


Carolyn Mayer


Uzoma Onunkwo


Cindy Phillips


Jared Saia (UNM)


Tim Shead
Cicada-mpc main author

Outline

- Application driver: Privacy-Preserving Machine Learning
- Algorithmic case study: dense matrix multiplication
- Software overview: Cicada-mpc (Fault-tolerant, open-source)

https://github.com/cicada-mpc/cicada-mpc/
https://cicada-mpc.readthedocs.io/

Secure MultiParty Computation

Example:

Secure Multiparty Computation Goes Live. Bogetoft et al. (2009)

Related work for Machine Learning:

- *SecureML: A System for Scalable Privacy-Preserving Machine Learning.* Mohassel and Zhang. (2017).
- *ABY^3: A Mixed Protocol Framework for Machine Learning.* Mohassel and Rindal. (2018)
- Many others (e.g. FALCON) for 2, 3, or 4 players.

Size of circuit for ML using traditional MPC approaches (e.g. EMP) is prohibitive.

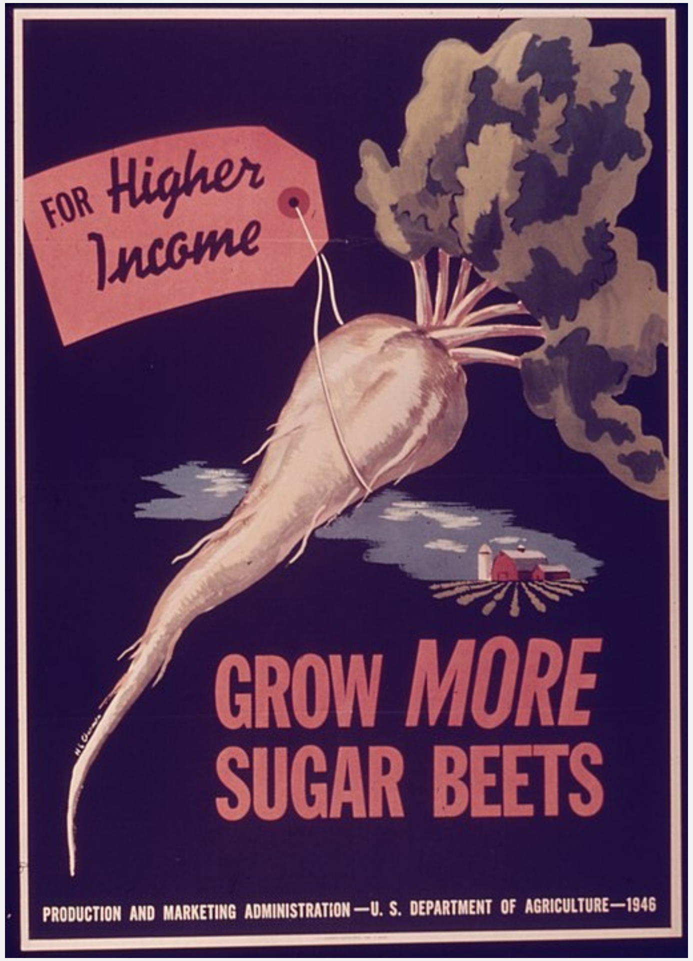


Image from National Archives. ARC Identifier: 514423

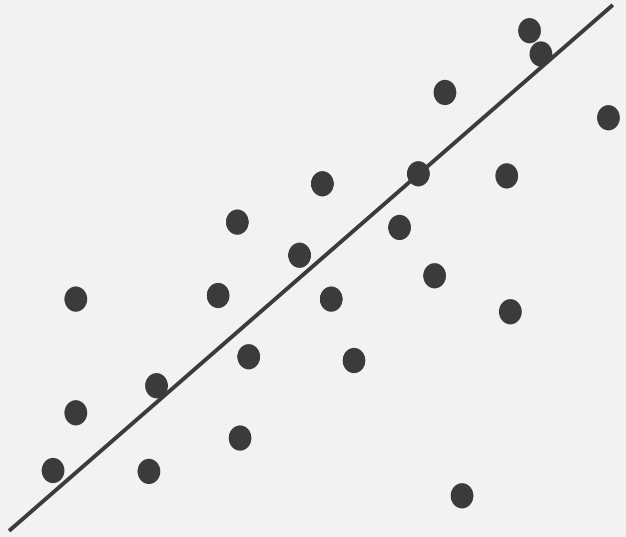
Motivation: MPC Linear Regression & Gradient Descent

Gradient descent:

Model: vector $\boldsymbol{\beta}$.

Goal: Minimize a loss function $\boldsymbol{L}(\boldsymbol{\beta})$ by iterating $\boldsymbol{\beta}' = \boldsymbol{\beta} - \eta \nabla \boldsymbol{L}(\boldsymbol{\beta})$

for some learning rate $\boldsymbol{\eta}$.



$\boldsymbol{\beta} = (m, b)$

Why linear regression?

- Single matrix-vector multiplication in each step.
- Allows for local computations.
- Hold shares of updated model locally.

Local Gradient Matrices

Global gradient G uses all datapoints.

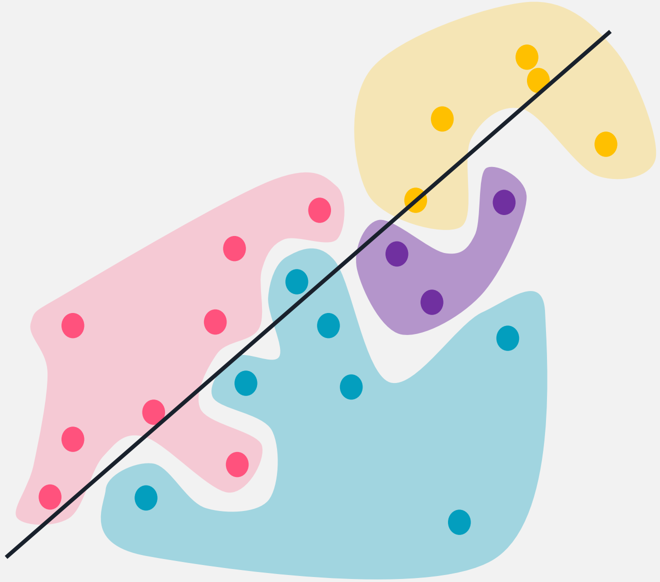
Local gradient G_p uses datapoints held by player p.

Then $G = \sum_p G_p$. Each G_p is a share of G.

Note: Players can have different amounts of data.

Note: Players' original gradient G_p is additive, ***but not shareable***

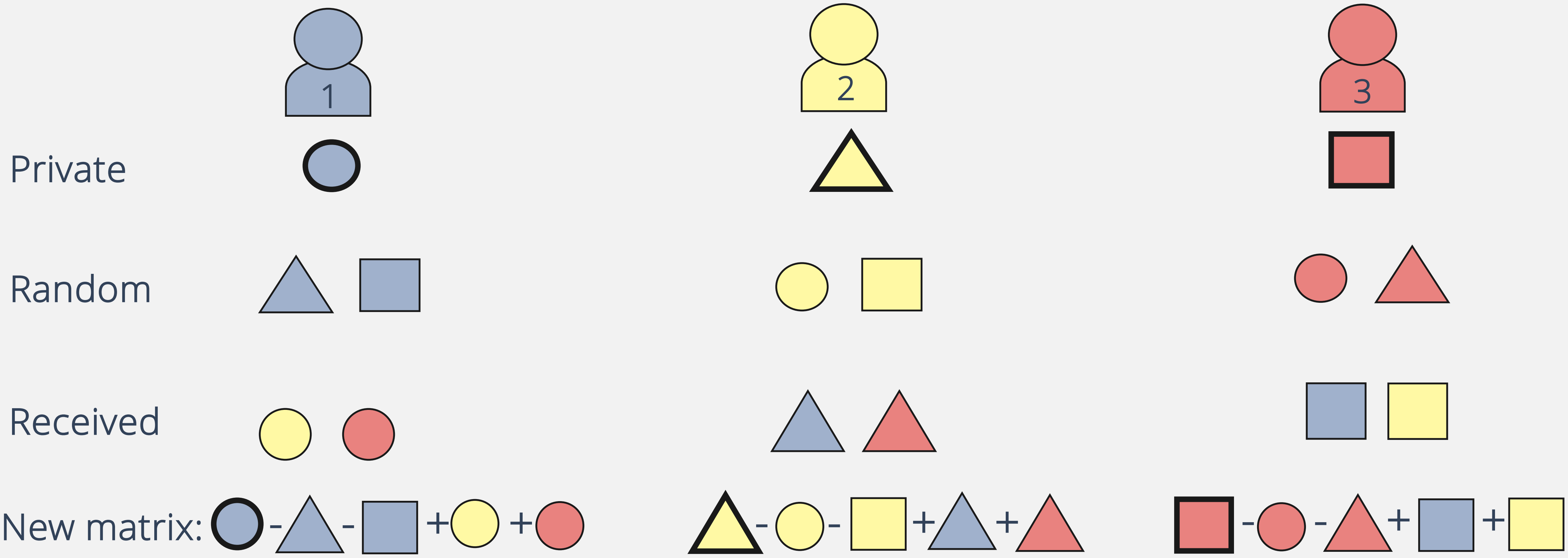
- *We create "additive secret shares" that are shareable*



$\boldsymbol{\beta} = (m, b)$

Typical MPC Computation: Resharing Matrices

Reshare to form matrices that don't individually reveal gradient information.



Private – Random + Received

MMULT(A, B)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

For each player p:

1. $A_p' \leftarrow$ AGGREGATE(A_p, C_p). # sum shares along columns
2. $B_p' \leftarrow$ AGGREGATE($B_p, \mathcal{R}_p$). # sum shares along rows
3. Return $A_p' B_p'$.

Where C_p is the column p is in and $\mathcal{R}_p$ is the row p is in.

Coalition resisted: $\sqrt{\#\ Players} - 1$

MMULT Example: 9 Players

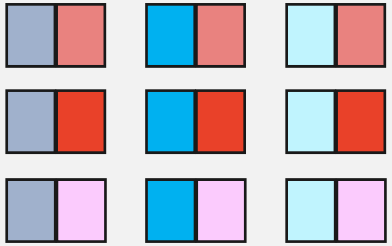
Aggregrate A in columns:

1	2	3
4	5	6
7	8	9

Aggregrate B in rows:

1	2	3
4	5	6
7	8	9

Local multiplications:



Global impact of MMULT:

$$\underbrace{(A_1 + A_4 + A_7)(B_1 + B_2 + B_3)}_{p_1} + \underbrace{(A_2 + A_5 + A_8)(B_1 + B_2 + B_3)}_{p_2} + \underbrace{(A_3 + A_6 + A_9)(B_1 + B_2 + B_3)}_{p_3} +$$

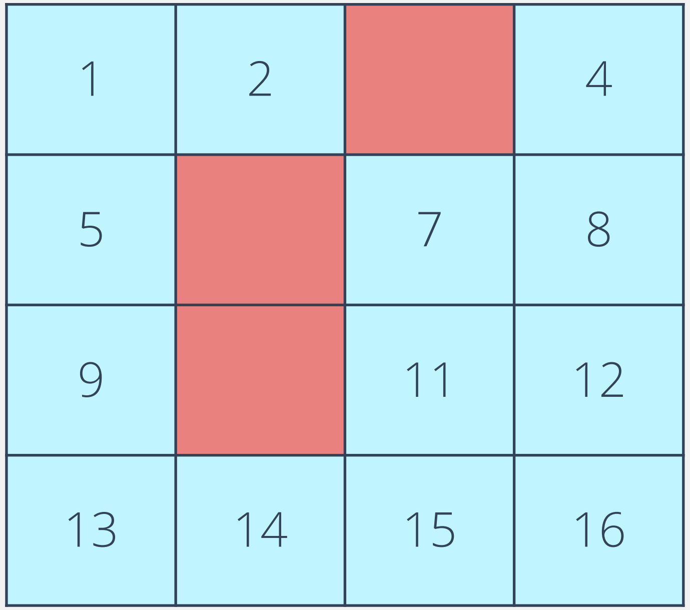
$$\underbrace{(A_1 + A_4 + A_7)(B_4 + B_5 + B_6)}_{p_4} + \underbrace{(A_2 + A_5 + A_8)(B_4 + B_5 + B_6)}_{p_5} + \underbrace{(A_3 + A_6 + A_9)(B_4 + B_5 + B_6)}_{p_6} +$$

$$\underbrace{(A_1 + A_4 + A_7)(B_7 + B_8 + B_9)}_{p_7} + \underbrace{(A_2 + A_5 + A_8)(B_7 + B_8 + B_9)}_{p_8} + \underbrace{(A_3 + A_6 + A_9)(B_7 + B_8 + B_9)}_{p_9}$$

Tolerating Fail-Stop Faults

Idea:

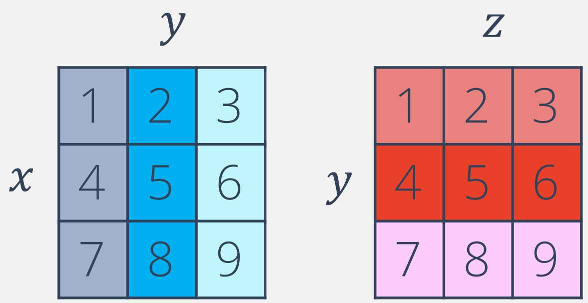
- Checkpoint row and column aggregated values.
- Use Cicada's built-in fault tolerance and Python exception handling



MMULT: Theoretical Results

The Communication Complexity (CC) of MMULT is *nearly optimal* for a single matrix multiplication, and ***optimal*** in the amortized sense for a suite of O($\sqrt{n}$) matrix multiplications (n is the number of players)

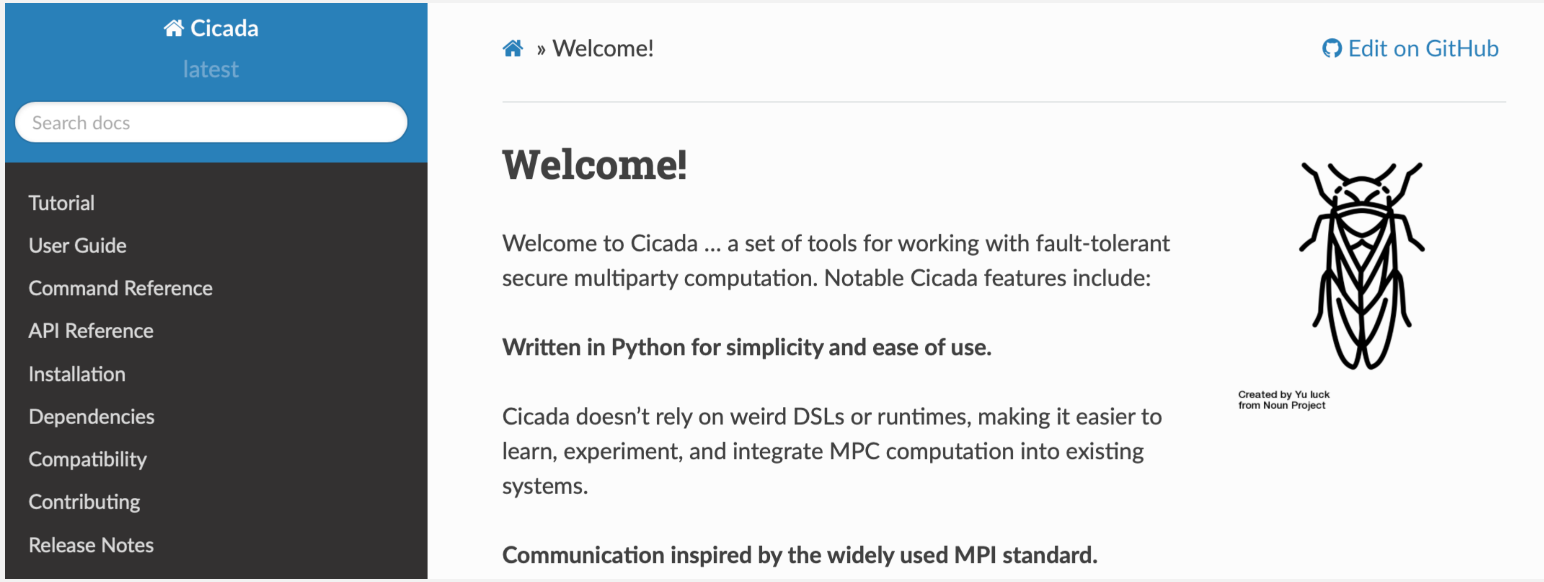
Method	Amortized CC	CC	Coalition Res.	Fail-stop Tol. (GD)
Shamir	$O(nxz)$	$O(nxz)$	$k-1 \leq n/2$	$n-k$
MMULT	$O(xy+yz)$	$O(\sqrt{n}(xy+yz))$	$\lceil\sqrt{n}\rceil - 3$	$\sqrt{n}-2$



CICADA Software Framework

- MPC software toolkit tolerating dropouts
- Open-source:
  - https://github.com/cicada-mpc/cicada-mpc/
  - https://cicada-mpc.readthedocs.io/

Cooperatlve Computing for Autonomous DAta centers



Written in Python, no weird DSLs or runtimes:

```
from cicada.communicator import SocketCommunicator

with SocketCommunicator.connect() as comm:
    print(f"Hello from player {comm.rank}!")
```

```
$ cicada run hello.py
Hello from player 0!
Hello from player 2!
Hello from player 1!
```

Based on three fundamental concepts

Communicators

Network abstraction representing an unchanging group of players, and communication patterns to pass messages among them.

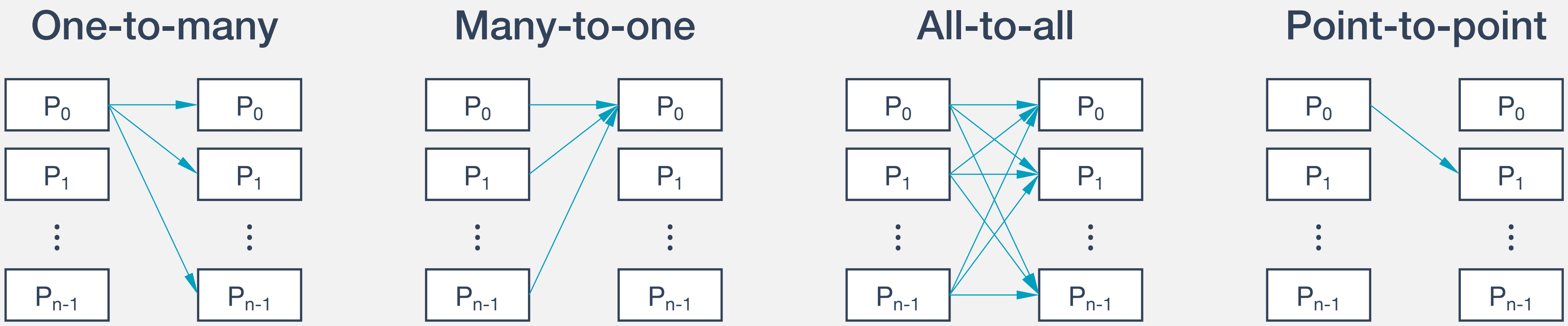
Encodings

Map between domain values and MPC-friendly integer field representations.

Protocol Suites

Use communicators and encodings to implement curated collections of privacy-preserving protocols: secret sharing, addition, multiplication, logical comparison, etc.

Communication Patterns



Based on three fundamental concepts:

Communicators

Network abstraction representing an unchanging group of players, and communication patterns to pass messages among them.

Encodings

Map between domain values and MPC-friendly integer field representations.

Protocol Suites

Use communicators and encodings to implement curated collections of privacy-preserving protocols: secret sharing, addition, multiplication, logical comparison, etc.

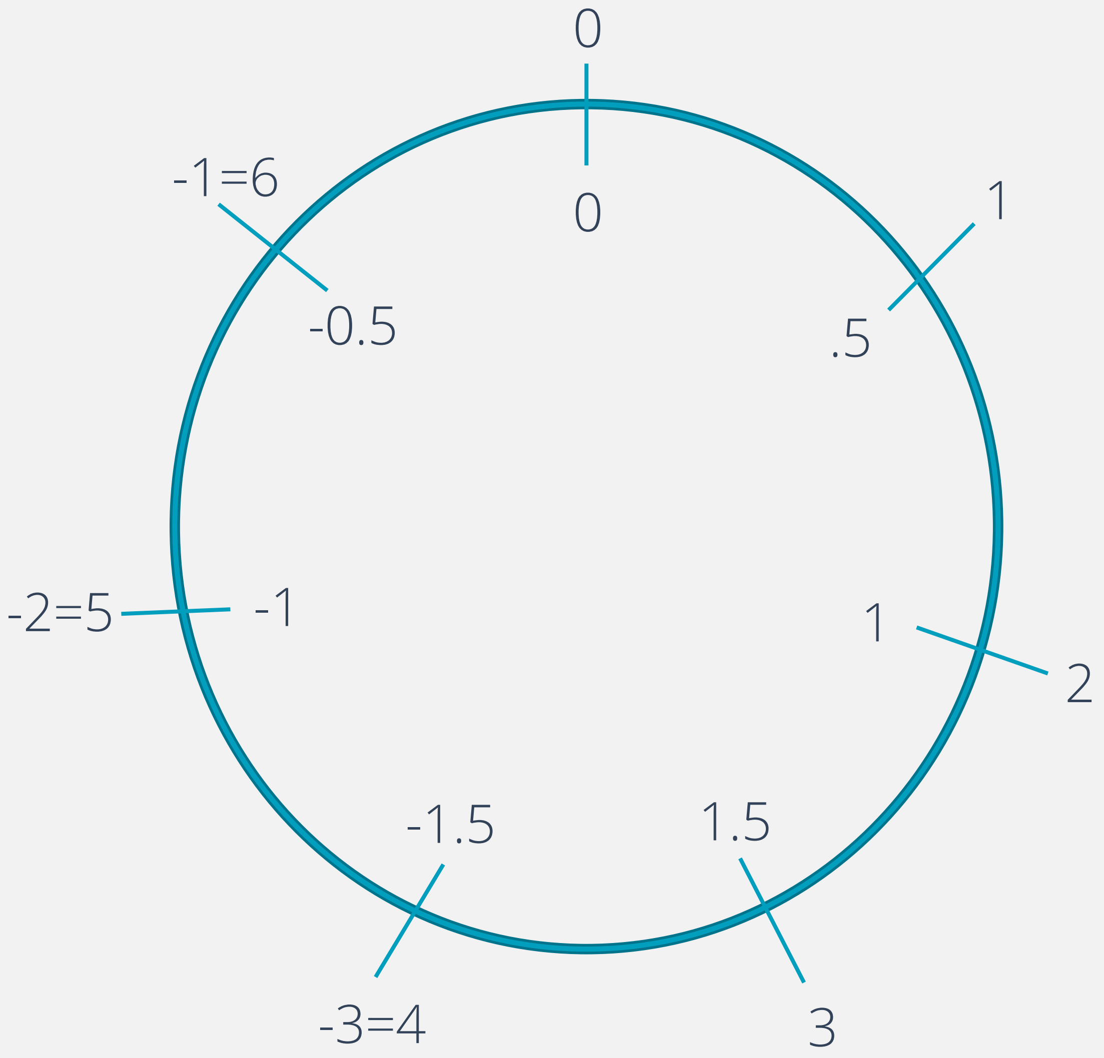
Encoding Fixed Point Arithmetic into a Field

Use fixed number of bits and two's complement arithmetic.

Lower order bits represent fractional part.

Example: 7-element field with lowest order bit representing fractional part.

Going forward, we use fixed point arithmetic in a field $\boldsymbol{F}$ with a prime number of elements.



Based on three fundamental concepts:

Communicators

Network abstraction representing an unchanging group of players, and communication patterns to pass messages among them.

Encodings

Map between domain values and MPC-friendly integer field representations.

Protocol Suites

Use communicators and encodings to implement curated collections of privacy-preserving protocols: secret sharing, addition, multiplication, logical comparison, etc.

The Millionaires' Dilemma in ~20 Lines of Cicada

```
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

    print(f"Winner: player {winner}")
```

Communicators

```
import numpy


from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

    print(f"Winner: player {winner}")
```

Encodings

```
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

    print(f"Winner: player {winner}")
```

Protocol Suites

```
import numpy

from cicada.additive import AdditiveProtocolSuite
from cicada.communicator import SocketCommunicator
from cicada.encoding import Boolean
from cicada.interactive import secret_input

with SocketCommunicator.connect(startup_timeout=300) as communicator:
    protocol = AdditiveProtocolSuite(communicator)

    winner = None
    winning_share = protocol.share(src=0, secret=numpy.array(0), shape=())

    for rank in communicator.ranks:
        prompt = f"Player {communicator.rank} fortune: "
        fortune = secret_input(communicator=communicator, src=rank, prompt=prompt)
        fortune_share = protocol.share(src=rank, secret=fortune, shape=())
        less_share = protocol.less(fortune_share, winning_share)
        less = protocol.reveal(less_share, encoding=Boolean())
        if not less:
            winner = rank
            winning_share = fortune_share

    print(f"Winner: player {winner}")
```

```
hostA $ cicada start --rank 0 millionaires.py

Player 0 fortune: 1230000
INFO:root:Winner: player 1
```

```
hostB $ cicada start --rank 1 millionaires.py

Player 1 fortune: 4560000
INFO:root:Winner: player 1
```

```
hostC $ cicada start --rank 2 millionaires.py

Player 2 fortune: 3400000
INFO:root:Winner: player 1
```

Fault Tolerance

Cicada is the only MPC library we're aware of with support for fault tolerance and recovery!

All communication patterns have explicit, finite timeouts ...

… so failures cannot go unnoticed.

Communicators raise exceptions when failures occur …

… this is the part where other MPC tools just die.

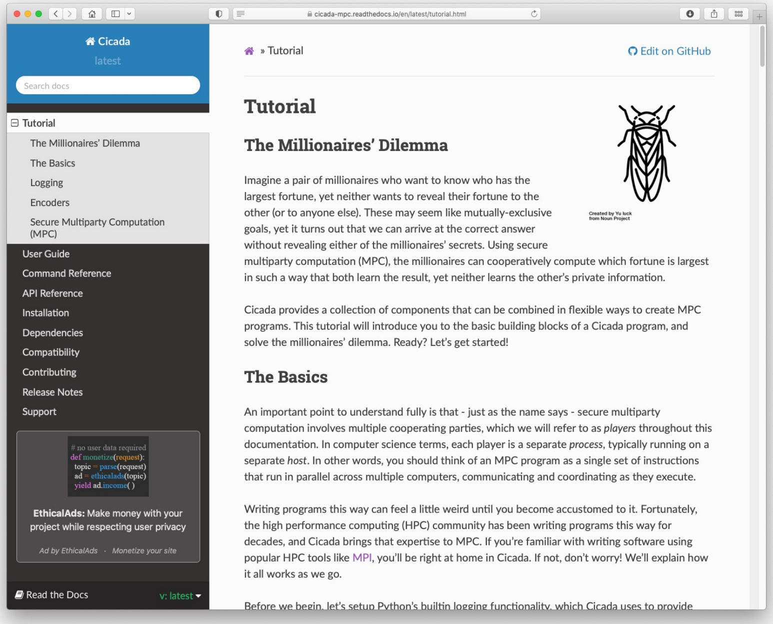
Applications can respond to exceptions in flexible ways …

… communicators can be *revoked* (preventing subsequent use by any player)

… communicators can be *shrunk* (returns a new communicator with the remaining players)

… data recovery is application specific.

Thorough Documentation


cicada-mpc.readthedocs.io/en/latest/tutorial.html

Cicada
latest
Search docs

Tutorial
The Millionaires' Dilemma
The Basics
Logging
Encoders
Secure Multiparty Computation (MPC)
User Guide
Command Reference
API Reference
Installation
Dependencies
Compatibility
Contributing
Release Notes
Support

EthicalAds: Make money with your project while respecting user privacy
Ad by EthicalAds · Monetize your site

Read the Docs v: latest

» Tutorial Edit on GitHub

Tutorial

The Millionaires' Dilemma

Created by Yu luck from Noun Project

Imagine a pair of millionaires who want to know who has the largest fortune, yet neither wants to reveal their fortune to the other (or to anyone else). These may seem like mutually-exclusive goals, yet it turns out that we can arrive at the correct answer without revealing either of the millionaires' secrets. Using secure multiparty computation (MPC), the millionaires can cooperatively compute which fortune is largest in such a way that both learn the result, yet neither learns the other's private information.

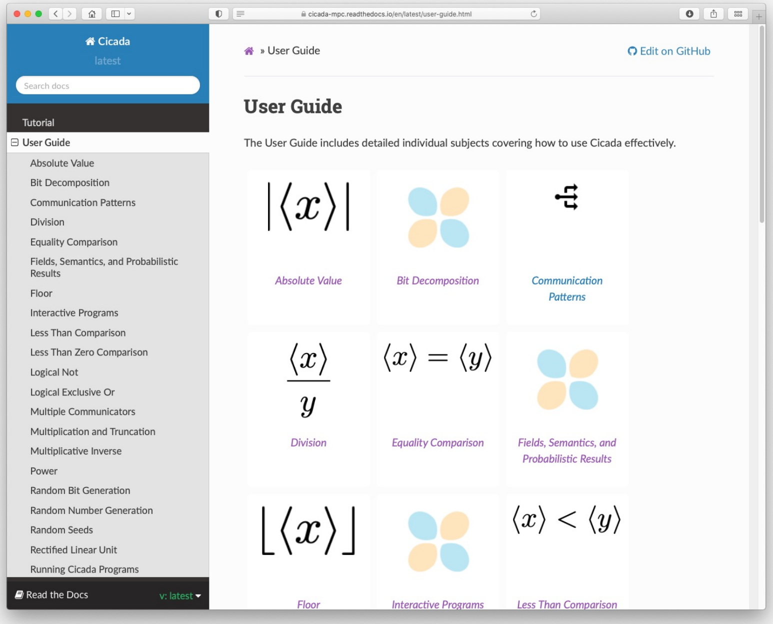
Cicada provides a collection of components that can be combined in flexible ways to create MPC programs. This tutorial will introduce you to the basic building blocks of a Cicada program, and solve the millionaires' dilemma. Ready? Let's get started!

The Basics

An important point to understand fully is that - just as the name says - secure multiparty computation involves multiple cooperating parties, which we will refer to as *players* throughout this documentation. In computer science terms, each player is a separate *process*, typically running on a separate *host*. In other words, you should think of an MPC program as a single set of instructions that run in parallel across multiple computers, communicating and coordinating as they execute.

Writing programs this way can feel a little weird until you become accustomed to it. Fortunately, the high performance computing (HPC) community has been writing programs this way for decades, and Cicada brings that expertise to MPC. If you're familiar with writing software using popular HPC tools like MPI, you'll be right at home in Cicada. If not, don't worry! We'll explain how it all works as we go.

Before we begin, let's setup Python's builtin logging functionality, which Cicada uses to provide


cicada-mpc.readthedocs.io/en/latest/user-guide.html

Cicada
latest
Search docs

Tutorial
User Guide
Absolute Value
Bit Decomposition
Communication Patterns
Division
Equality Comparison
Fields, Semantics, and Probabilistic Results
Floor
Interactive Programs
Less Than Comparison
Less Than Zero Comparison
Logical Not
Logical Exclusive Or
Multiple Communicators
Multiplication and Truncation
Multiplicative Inverse
Power
Random Bit Generation
Random Number Generation
Random Seeds
Rectified Linear Unit
Running Cicada Programs

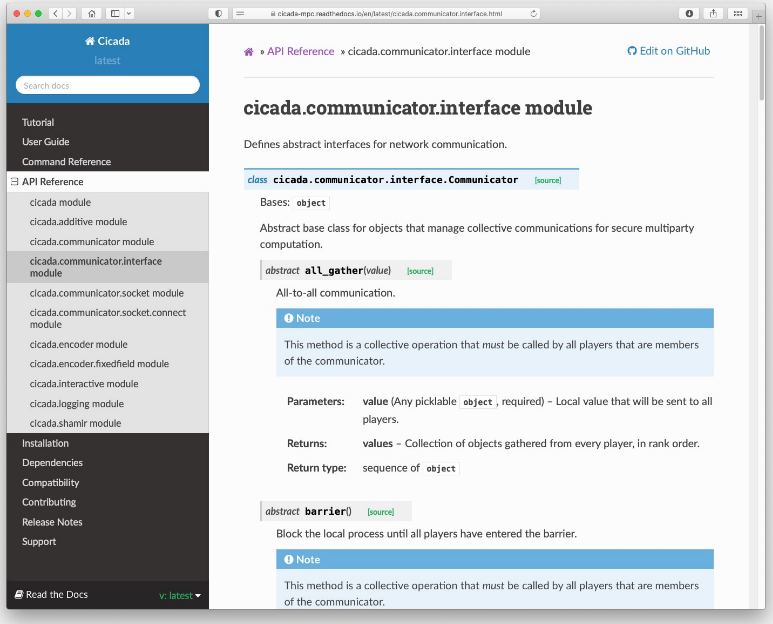
Read the Docs v: latest

» User Guide Edit on GitHub

User Guide

The User Guide includes detailed individual subjects covering how to use Cicada effectively.

$|\langle x \rangle|$ Absolute Value
Bit Decomposition
Communication Patterns
$\frac{\langle x \rangle}{y}$ Division
$\langle x \rangle = \langle y \rangle$ Equality Comparison
Fields, Semantics, and Probabilistic Results
$\lfloor \langle x \rangle \rfloor$ Floor
Interactive Programs
$\langle x \rangle < \langle y \rangle$ Less Than Comparison


cicada-mpc.readthedocs.io/en/latest/cicada.communicator.interface.html

Cicada
latest
Search docs

Tutorial
User Guide
Command Reference
API Reference
cicada module
cicada.additive module
cicada.communicator module
cicada.communicator.interface module
cicada.communicator.socket module
cicada.communicator.socket.connect module
cicada.encoder module
cicada.encoder.fixedfield module
cicada.interactive module
cicada.logging module
cicada.shamir module
Installation
Dependencies
Compatibility
Contributing
Release Notes
Support

Read the Docs v: latest

» API Reference » cicada.communicator.interface module Edit on GitHub

cicada.communicator.interface module

Defines abstract interfaces for network communication.

class `cicada.communicator.interface.Communicator` [source]

Bases: `object`

Abstract base class for objects that manage collective communications for secure multiparty computation.

abstract `all_gather`(*value*) [source]

All-to-all communication.

Note

This method is a collective operation that *must* be called by all players that are members of the communicator.

Parameters: **value** (Any picklable `object`, required) – Local value that will be sent to all players.

Returns: **values** – Collection of objects gathered from every player, in rank order.

Return type: sequence of `object`

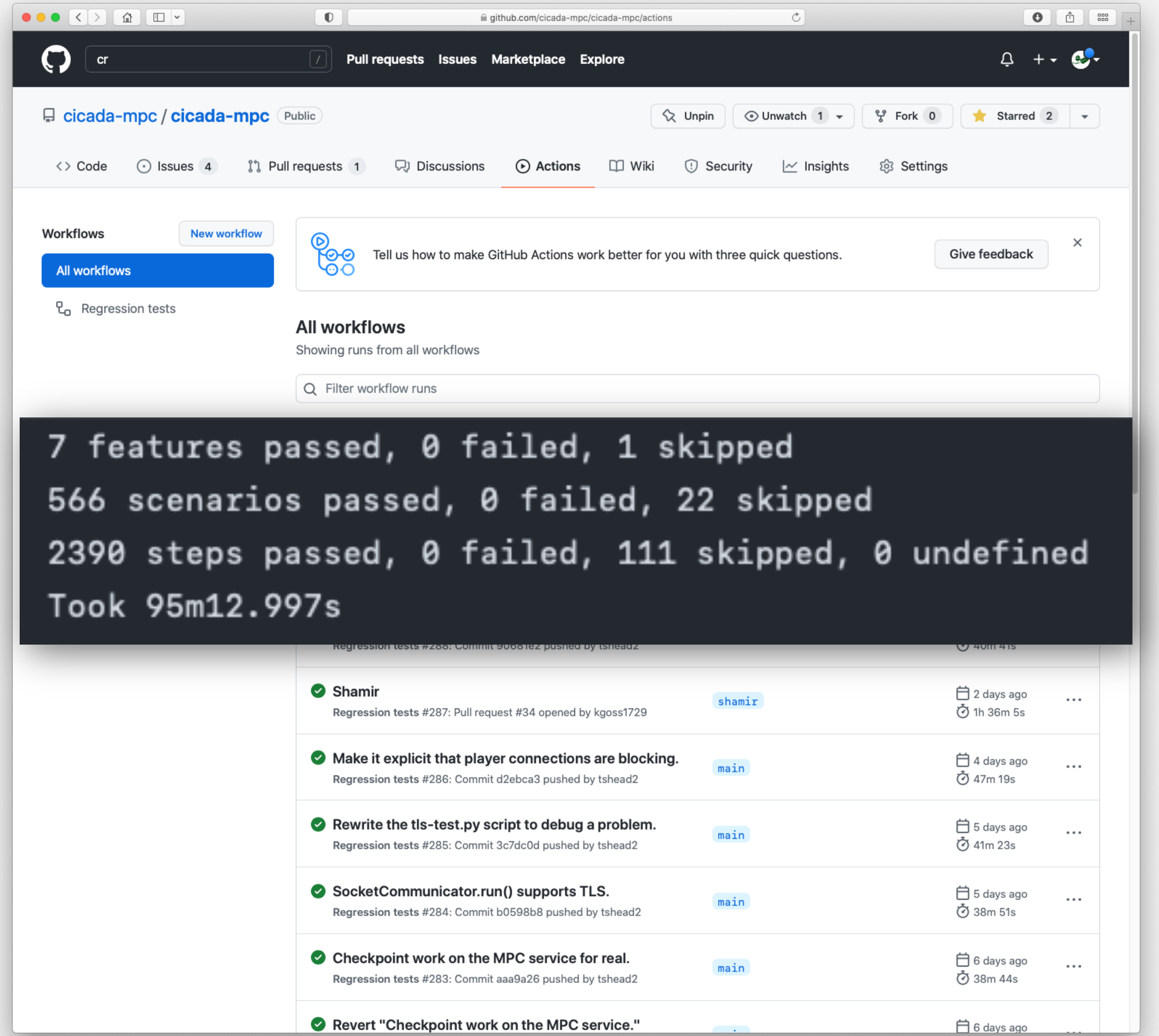
abstract `barrier`() [source]

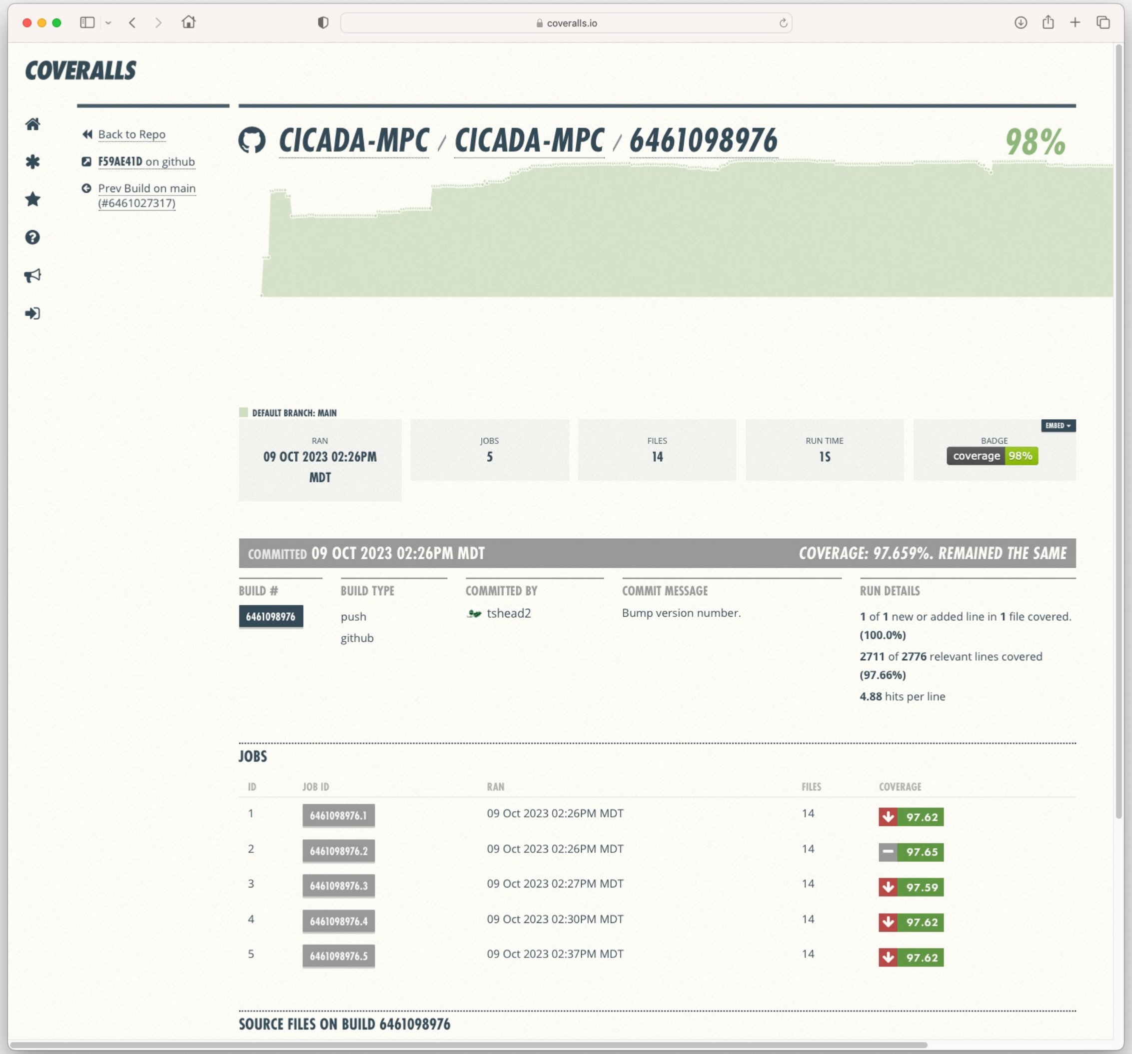
Block the local process until all players have entered the barrier.

Note

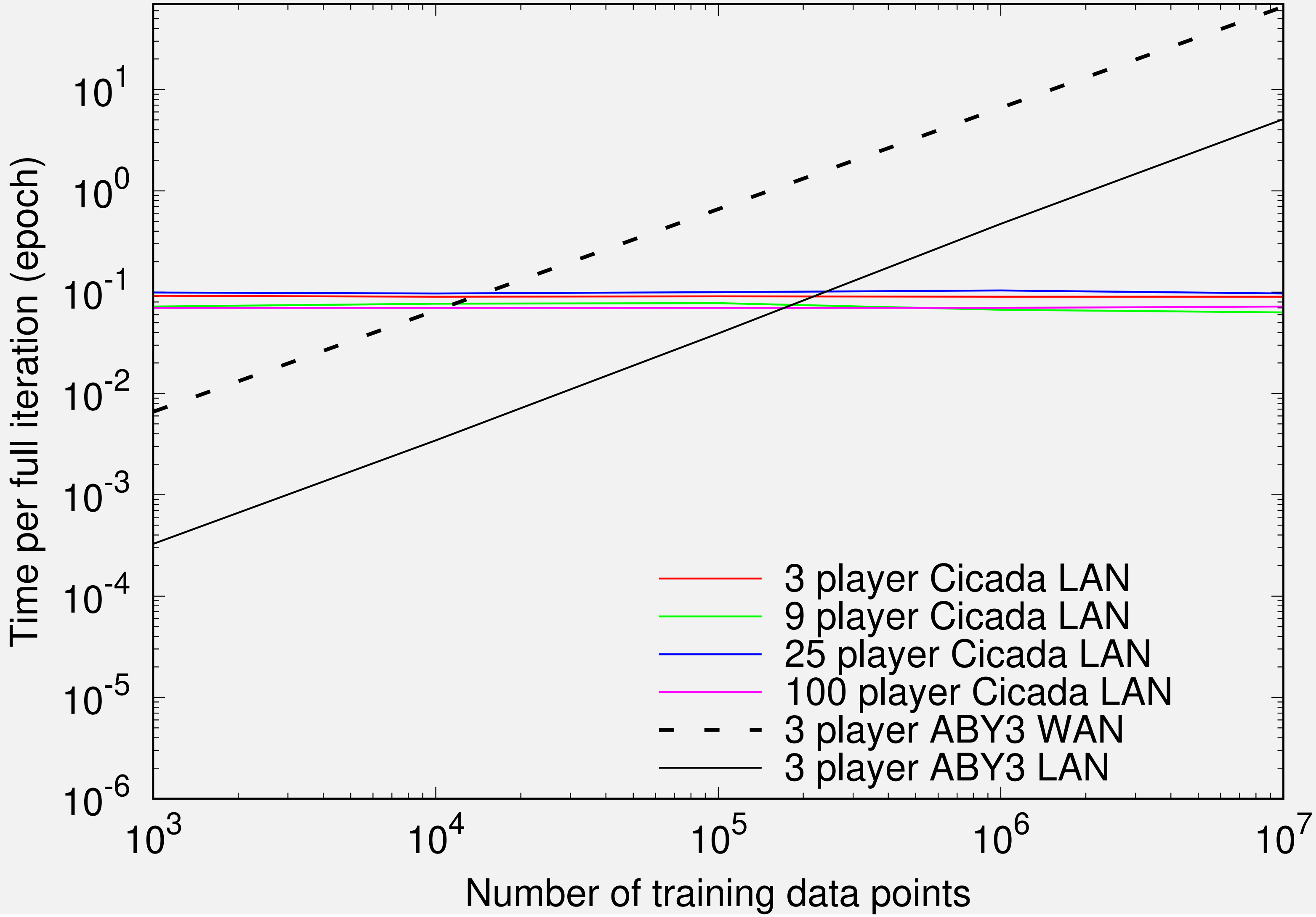
This method is a collective operation that *must* be called by all players that are members of the communicator.

Thorough Testing and Continuous Integration





MPC Through 100 Players!



Conclusions, HPC Community Asks

"WHY DIDN'T WE USE MPI and USER-LEVEL FAULT MITIGATION (ULFM)?"

Three years ago, we evaluated ULFM reference implementations in MPICH and OpenMPI. We identified problems such as:

- Communicator revocation wasn't detected by all ranks, depending on which ranks initiated the revocation.
- Some collective operations did not raise timeout errors even when some ranks were dead.
- Because ULFM hasn't been adopted by MPI, the Python mpi4py bindings don't support ULFM, and working with patched bindings severely limits our ability to distribute our software.

Questions?

jberry@sandia.gov